ZXzine

Issue #6

November 2018

Special Timex Sinclair 2068 Issue

ZXzine

Published by:
Timothy Swenson
swenson_t@sbcglobal.net
swensont@lanset.com

ZXzine is published as a service to the Sinclair ZX81 community. Writers are invited to submit articles for publication. Readers are invited to submit article ideas.

Created using Open Source Tools:

- OpenOffice
- Scribus
- Gimp
- SZ81
- EightyOne

Table of Contents

Editorial **1**

Timex Sinclair 2068 History **1**

T/S 2068 Specifications **3**

Basic Tools for T/S 2068 Emulators **4**

Basic Compilers for the T/S 2068 .... **5**

T/S 2068 Emulators **5**

T/S 2068 Assembly Language Programming **6**

Loading Screens **9**

ZXpaintyone **10**

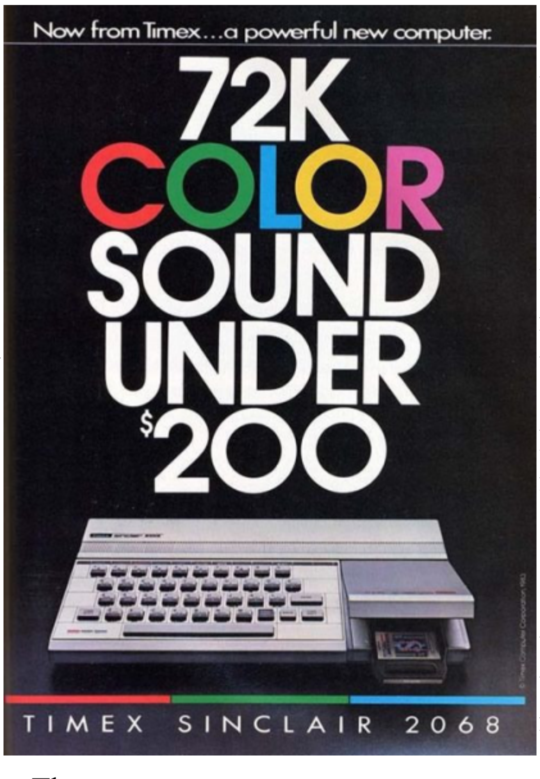
QDDASM - Quick & Dirty DisAssembler **11**

Editorial

This issue is focused on the Timex/Sinclair 2068. I was a subscriber to Sync magazine, the only magazine that was covering the Sinclair line of computers in the US, when the Spectrum was announced. Due to the lead time of magazines, I read about the Spectrum in the summer of 1982, months after it was released in the UK. It was then a wait to see when it would be introduced into the United States.

The specifications on the Spectrum were not ground breaking. The Atari 400/800 and the Tandy Color Computer had about the same graphics capability (everyone judged the computer by the gee-wiz graphics on those days). What was groundbreaking, of course, was the price. Sinclair was known for coming in with a low price and the Spectrum was far cheaper that its competition.

Then there was the long wait for the Spectrum to be released in the American market. Instead of releasing the Spectrum, Timex created the T/S 2068. When the T/S 2068 was released, despite starving student in college, I bought one as soon as it hit the local stores. With Timex watches being available in most drug stores, the T/S 2068 was available at Payless Drugs, Longs Drugs, and even some catalog stores. Of course, no employees at the store had any idea how to use the computer, it was just a box to sell.



I put my ZX81 aside and started on the T/S 2068. With limited information (mostly just the User Guide) I tinkered and tinkered and took the T/S 2068 for a test drive. Then bad luck struck. Timex called it quits and the T/S 2068 was taken off the market. Stores sold their stock. A few months after this, my T/S 2068 started failing. It was too late to get a replacement. There was no local computer shop that would repair it (I tried). So, into the closet it went, never to be used again.

A few years later, I moved up to the QL. I had fond memories of the time I spend with the ZX81, but very little memories of the T/S 2068. When I joined Timex/Sinclair clubs and was exposed to the T/S 2068, I was not interested as I had a QL that was, to me, a real computer.

Only now, many years later am I looking to explore what I did not explore earlier with the T/S 2068. I want to explore those parts to the T/S 2068 that make it different that the Spectrum. My main focus will be on assembly language programming. As the T/S 2068 is similar to the Spectrum, there are enough differences that does not let Spectrum assembly programs work on the T/S 2068.

I have not abandoned the ZX81 and there are a few interesting ZX81 oriented articles.

Timex Sinclair 2068 History

When Timex created the Timex Sinclair 1000, it had made an agreement with Sinclair that it was the sole company that could market Sinclair products in the United States. Originally Sinclair sold the ZX81 through mail order in the United States, but Timex decided that they would flood the US with the ZX81 through all the stores that already carried Timex products. So, the T/S 1000 came out and it really was available everywhere.

When the Spectrum was released in the UK, Sinclair had to let Timex do the introduction into the United States. Timex had originally planned to make minor changes on the Spectrum and sell it in the United States. Then Timex hit the FCC. When going through certification with the FCC, the changes necessary to meet the FCC standards required a whole new chip to be designed and built.

With the delay in creating the new chip, Timex decided to add some new features. In the fast moving home computer market, a computer could be outdated in 6 months to a year. Timex wanted to make the computer more usable for games, so they added a cartridge port. To challenge other home computers Timex added the AY sound chip. With the cartridge port, Timex also added the idea of bank switching to address more memory. These changes would later greatly affect the users.

At the Boston Computer Society, on October 26, 1983, a year and a half after the Spectrum was introduced in the UK, Timex introduced the Timex Sinclair 2000. The "2000" was the working title of the computer and was natural sequence after the T/S 1000. If I remember it right, Timex wanted to release the T/S 2000 with different memory versions, so the T/S 2068 was named, and then the changed their mind and just stayed with the 2068.

The T/S 2068 was released to a fairly crowded home computer market, mostly dominated by US computer companies like Commodore, Atari, Texas Instruments, Tandy, and even Mattel. American consumers liked to buy what they saw advertised and what they saw support for. Anyone walking into a computer shop in 1983 did not see a Sinclair or Timex Sinclair. There was little support from the computer stores and no support from the stores that sold the Timex/Sinclair computers. You could walk into a retail store, buy a T/S 2068 and a microwave, and you would not expect the staff to help you use either one of them.

After being on the market for 6 months, Timex called it quits. For anyone taking the time to think about what computer to purchase, that was a signal to not buy a T/S 2068. A number did buy them as the stores were closing out their stocks, but I think the vast majority were mostly retired within a year, once the owners found that it took some work to understand a home computer.

The T/S 2068 retailed for $199.99. The price did not seriously drop when Timex left the market. The price for the T/S 1000 did get to a closeout of $20 or less, but the T/S 2068 did not really drop.

The T/S 2068 community did continue through the local user groups and a few Timex Sinclair newsletters and low-cost magazines. When Timex left the market the two glossy magazines that covered T/S computers, closed shop. A few small businesses continued support for the T/S 2068. Software was written, hardware was designed, and users found that the T/S 2068 was popular at annual Timex/Sinclar Fests held around the US.

1983 was the year of the home computer wars in the United States. The casualties were Texas Instruments, Mattel, Timex, and Coleco. Even more established brands removed some models from the market. For Commodore the C16 and Plus4. For Tandy the MC-10.

After Timex left the market, the T/S 2068 still had enough of a user base that a number of small companies popped up to support it. It soon took over the Sinclair user groups as the most popular computer. The same happened with the different Sinclair newsletters.

There was new hardware developed for it, including a number of disk systems. Larken, Oliger, Aerco, and Zebra made disk interfaces. From reading the newsletters, Larken and Oliger were the most popular. Larken also made a Ramdisk, so that files could be saved quickly, but later had to be copied off.

Software continued to be written. There were a few T/S 2068 programmers that really knew the computer and were able to produce some good software. Pixel Print and its companion tools like Icon Package and Font Package, moved the T/S 2068 onto the world of desktop publishing. Some users explored the bank switching feature of the T/S 2068.

Aerco developed a cartridge that supported the higher resolution screens and allowing 64 characters per line. A BASIC toolkit, Window Print 64, was created to allow window and print tricks for this mode.

There were at least two Spectrum ROM add-ons made to allow Spectrum software to run on the T/S 2068. One was an internal ROM that had an external switch that selected T/S 2068 ROM or the Spectrum ROM. Another was a Cartridge that fit in the Dock.

RMG Enterprises of Oregon carried a lot of hardware and software for the T/S 2068 and specialized in finding T/S 2068 program authors and publishing their software.

Time Designs and Update! magazines supported the T/S 2068. Time Design closed in the late 1980s and Update continued until at least the mid-1990s. It was about the mid-90's that most dealers for the T/S 2068 started shutting down and user groups were starting to fade.

T/S 2068 Specifications

The T/S 2068, although based on the Spectrum, it is not 100% compatible with the Spectrum. The major changes are:

1. Adding Cartridge Dock

On the right side to the T/S 2068 is a door that opens up to reveal a cartridge dock. This allowed youth to use the computer more like a video game system and not taking the time to load games from tape. This dock required the hardware to change and support bank switching. The dock can also be used to install a board with a Spectrum ROM and allow the T/S 2068 to run Spectrum software.

2. AY Sound Chip

The T/S 2068 supports the traditional Spectrum BEEP command for sound, but to expand on that, an AY sound chip was added. This is the same chip that is in the Zon X-81 sound system for the ZX81 and the Spectrum 128.

4. Additional Basic Commands

There are a number of new commands in T/S 2068 Basic that are not on the Spectrum. Examples are FREE, STICK, SOUND, DELETE, etc. The STICK command was for the built in joystick ports and the SOUND command is for the AY sound chip.

5. Additional Video Modes

The T/S 2068 supports some additional video modes; Display Mode 2 (64 Columns), Display Mode 3 (two screens), and Display Mode 4 (Ultra High Color Resolution mode). These modes are really not used and none of the guides are books explain how to use them. The exception is a OS64 cartridge that allows the use of the 64 column node with BASIC.

6. Expansion Port Changes

The expansion port on the back of the T/S 2068 was changed and it not compatible with the Spectrum, so any Spectrum interfaces will not work with the T/S 2068 without an adapter.

7. Memory Map Changes

With the hardware changes and the additional display modes, there were changes in the memory map of the T/S 2068.

8. ROM Changes

With all of the above changes, the ROM had to change. The ROM has the routines pretty much in the same order as the Spectrum, but the addresses of the ROM routines have changed, so any Spectrum programs that used machine code will fail on the T/S 2068.

9. Built in Joystick Ports

The T/S 2068 has two built-in D style joystick interfaces that worked the same as most other American computers joysticks, which allowed the common Atari joystick to be used. A built-in joystick interface would not be included on the Spectrum until the +2.

The changes made to the T/S 2068 would have worked, if Timex had remained in business and created the software to support all of these additional features, but in the end it just meant that T/S 2068 users really could not use any of the Spectrum software from the huge market in the UK. A few programs were ported to the T/S 2068, but not enough.

The Basic on the T/S 2068 only had minimal changes, so Basic programs written for the Spectrum would run on the T/S 2068. Even then, there was little effort put into publishing the Spectrum Basic books in the US.

Basic Tools for T/S 2068 Emulators

With the use of emulators, it is fairly easy to write code using an editor and have it converted into a TAP file that will work on the emulators.

Despite the few differences in Basic between the Spectrum and the T/S 2068, tools used for the Spectrum should work.

I've been tinkering with zmakebas. It comes as a C program that I was easily able to compile for my Linux system. Almost any TAP file that I create with zmakebas for the Spectrum will also work on the T/S 2068.

What I like about zmakebas is that it supports not using line numbers and using labels. It ignores empty lines in between code.

I do have to avoid using any of the Basic commands that are specific to the T/S 2068. If I needed them there are two options. One would be to add code to zmakebas to include those BASIC commands. The other option would be to write the code and use the equivalent character on the Spectrum as the command on the T/S 2068. The additional T/S 2068 commands replace Spectrum characters. The FREE command has the value of 126. On the Spectrum that is a tilde. If I put a tilde in Basic, it will show up on the T/S 2068 as the FREE keyword.

Another tool is bas2tap. It also comes with C source that I was able to compile. It will take a BASIC program witten in ASCII and convert to a TAP file. It is more restrictive than zmakebas as line numbers are needed. It has syntax checking, which can come in handy.

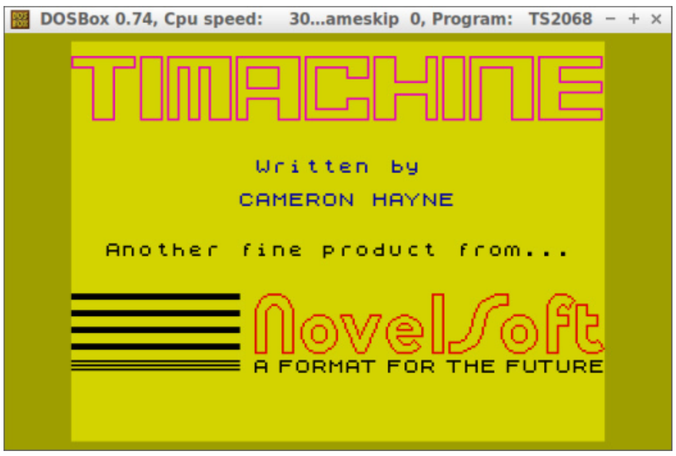
I've tested both tools and they both produce a TAP file that will work on the T/S 2068. My preference is to zmakebas, as I like not having to worry about line numbers.

Basic Compilers for the T/S 2068

There are two compilers for the T/S 2068, ZIP and Time Machine.

ZIP was written for the Spectrum by Simon Goodwin. Knighted Computers of New York licensed ZIP for the T/S 2068 and had Simon do the port. Simon, in a posting on the web, said that during the porting of ZIP he did not have access to a T/S 2068, but was given a list of ROM call locations. After some work, he was able to create a working compile for the T/S 2068.

I've looked online but I have not seen any archived copies of ZIP for the T/S 2068. The Spectrum version is not available for distribution and this could also affect the T/S 2068 version.


DOSBox 0.74, Cpu speed: 30...ameskip 0, Program: TS2068

TIMACHINE

Written by
CAMERON HAYNE

Another fine product from...

NovelSoft
A FORMAT FOR THE FUTURE

Time Machine was originally sold for the Spectrum as HiSoft Basic. Novelsoft had it ported to the T/S 2068 and renamed it Time Machine. Oddly enough, the load screen for Time Machine says "TIMACHINE".

I found Time Machine on a T/S 2068 archive site. It loads fine and works as described in the HiSoft Basic manual, but it has an issue. No matter what program I have loaded to compile, it does not compile. Once a program is compiled, the screen will show the size of the compiled program and the location in memory. In each case, the size has always been 0. I know that HiSoft Basic is a subset of Sinclair Basic, so I've kept my example programs small and easy, even one that was just a few print statements. They all compile to 0 bytes. I have tried Time Machine with both Warajevo and EightOne and have gotten the same results.

To make sure I was not doing something wrong, I stared EightyOne in Spectrum mode, loaded HiSoft Basic and compiled a simple program. That worked. It showed the number of bytes.

I'm not sure why Time Machine is not working.

T/S 2068 Emulators

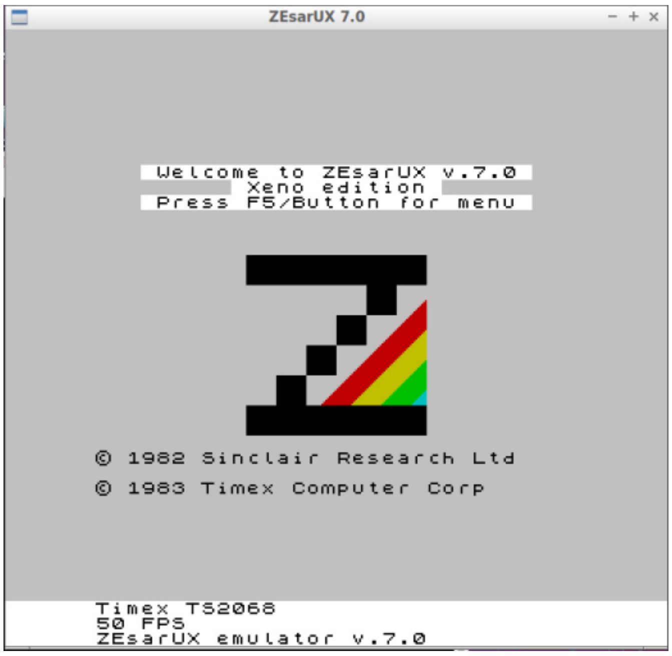
These days a lot of people are using emulators for retro computing. Real hardware is failing, it is harder to find and when it is found, some people want a lot of money for it. With emulators, these issues can be avoided. Plus, with cross development tools, it is easier to develop on a modern operating system and then port code to the emulated system.

There are three emulators that support the T/S 2068, EightyOne, Zesarux and Warajevo. EightyOne and Zesaurux emulators are multi-platform emulator, meaning that they emulate a number of computers including the entire Sinclair and Timex Sinclair line of computers. Warajevo is a DOS-based emulator that does the Spectrum (48K and 128K) and the T/S 2068.

EightyOne is Windows based and has been updated in the last year or two. It can be easily found via a

Google search. It is a simple executable so installation is trivial. EightyOne has been tested on Linux with WINE and to works just fine. The list of systems emulated is listed by company, Sincliar, Timex Sinclair, etc. You can emulate a ZX81 or a T/S 1000, which are pretty much the same system.

Being a Windows program there is a nice GUI used with the emulator. This makes it easy to pick tape files to load (.TAP) and Dock files to "plug in". EightyOne is probably one of the most popular emulators for Sinclair systems and should be known by almost everyone.

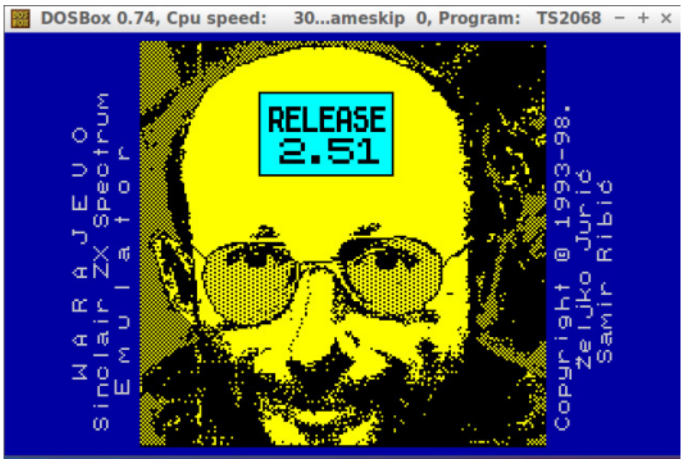


```
Welcome to ZEsarUX v.7.0
    Xeno edition
Press F5/Button for menu

© 1982 Sinclair Research Ltd
© 1983 Timex Computer Corp

Timex TS2068
50 FPS
ZEsarUX emulator v.7.0
```

Zesarux is Linux based, is very actively updated and the newest emulator on the block. Zesaurux comes with source code and is easily compiled. As the most recently updated emulator it does have the latest features. It has a mouse driven menu system that makes it easy to change options. The documentation is limited. The author is always posting to the different forums about the latest version and what fixes and new features that it has.

Warajevo is an older DOS-based emulator that works fine using DOS-BOX. DOS-BOX is a DOS emulator that will run on Windows and Linux. Warajevo was written in the late 1990's by two guys from Sarajevo. The emulator has a range of options detailed in a huge user guide. The user guide details sections on the emulation of the tape recorder, microdrives, network, MIDI interface, etc. There is even a built in monitor for doing assembly language programming.

Since it is DOS-based, there is no GUI around the emulator. Control of the emulator is done through function keys. The emulation seems to be a little slow, but that could be the emulator trying to emulate the speed of the T/S 2068



Despite how old Warajevo is, it is still a good emulator with lots of features. Using DOS-BOX makes it easy to run the emulator on a number of platforms. Once I figured out the function keys, using Warajevo was fairly easy.

Googling will find a few T/S 2068 archives. World of Spectrum has some software. Another is http://k1.spdns.de/Vintage/Sinclair/82. The software comes in .Z80, .TAP and .TZX. Cartridges come as .DCK. A fair bit of the software that Timex released is available. Most of the software available is games. I have found some software written by US authors Musicola, Multi-Draw, War in the East and Britain Invaded.

T/S 2068 Assembly Language Programming

I have done ZX81 assembly language programming for a few years. Now that I am getting into the T/S 2068, I thought I would learn what I could on programming the T/S 2068 in assembly.

I started by reading what I could on the T/S 2068. There in an Intermediate/Advance Guide to the T/S 2068 that has a section on machine code. Reading through that, there was little helpful information. There were a few short examples, but nothing more than the very basics.

I then trolled through some old T/S 2068 publications that I had, such as Time Designs, Update and some user group newsletters. I found some examples, but nothing too instructive.

There was also the issue of what assembler to use and how to get it to output a binary that will run on the T/S 2068. I've used Pasmo for the ZX81 so I looked into it. For the ZX81, there is a template that sets up the BASIC statements and REM statement.

Then I realized that the main different between the Spectrum and the T/S 2068 is the location of the ROM routines. If I could find some instructions on Spectrum assembly language, I just had to convert the ROM routine address from the Spectrum to the T/S 2068.

Here are the different documents that I found in my search to learn more about T/S 2068 assembly.

For the T/S 2068:

T/S 2068 Technical Manual - details the name of the ROM routines and their location.

Timex/Sinclair 2068 Intermediate/Advanced Guide by Jeff Mazur

"2068 BASIC ROM Calls", Ray Kingsley, SyncWare News

For the Spectrum:

Spectrum Complete ROM Disassembly, Dr. Ian Logan

Spectrum Machine Code Made Easy, James Walsh

Mastering Machine Code on your ZX Spectrum, Toni Baker

How to Write ZX Spectrum Games, Jonathan Cauldwell

I am using Pasmo as my assembler because it will create a .tap file for the Spectrum that has the BASIC lines to start the assembly program. Initially I was thinking that I had to create a .bin file of just the assembly code and use the LOAD .. CODE command to get it into the emulator, but reading the Pasmo documentation I found the --tapbas option.

I also created a file with all of the ROM routines so that I don't have to call them by number, but by name. I did the same thing for the system variables.

```
;; test1.asm
;;     Print a single 'A' character

ORG 65000

   LD A,65      ; put "A" char in A
   RST 16       ; print
   RET          ; return

END 65000
```

Test1

The first program I did was a simple "print a character to the screen". This was a simple as putting the character to print into the A register and then call RST 16 (or RST $10 in hex) to print to the screen.

I compiled the program like this:

```
pasmo --tapbas test1.asm test1.tap
```

The BASIC lines that Pasmo puts in autoruns the program, so it is run as soon as it is loaded into an emulator.

Instead of using a REM statement for storing the program, I have loaded it into upper memory at location 65000.

Test2

The second test program takes another baby step and adds a ROM call for CLS to clear the screen. In the previous example, the A character was printed at the next print position. After loading the program, there was text on the screen documenting the load, and the A was printed after that.

Test3

The third test program adds a loop to print a number of characters to the screen. Again, it is just another baby step.

Test4

In the fourth test program, instead of using RST 16 to print, the program copies the character directly to the memory location of the screen. Screen memory begins at 16384. This program uses the LDIR command as a form of loop.

Test5

The "Intermediate/Advanced Guide" book talks about the Function Dispatcher. This is a way to call ROM routines. It is very well documented in the T/S 2068 Technical Manual with the ROM routines documented as Service Routines access through the Function Dispatcher.

To use the Function Dispatcher, there is some set up code with the DE register pair, then the Service Routine number is loaded into DE and then the Function Dispatcher is called. In the test program, BC is used for storing an X and Y coordinate. Then the PLOT routine is called. The whole program is 11 lines of assembly. It seems a little too much just to use PLOT.

```
;; test5.asm
;;    Plot a point using the
Function Dispatcher

INCLUDE 'lib/rom.asm'

ORG 65000

          LD   BC,7FAFh
loop:     PUSH BC
          LD   DE,0000
          PUSH DE
          PUSH DE
          LD   DE,89
          PUSH DE
          CALL FUNCDISP
          POP  BC
          DJNZ loop
          RET

END 65000
```

Test6

This the same program as test 5, but I had called the PLOT ROM call directly. Well, technically I am calling the PLOTBC ROM call, where the X and Y coordinate are in the BC register pair. This program is 6 lines of assembly, saving 5 lines by not using the Function Dispatcher. I'm not sure why the Function Dispatcher was created as it seems like there is a lot of overhead when using it.

Test7

To test the SETAT or PRINT_AT ROM routine, this program was created. The print location (column and row) is put into BC and SETAT is called. Then RST 16 is called and the character is printed at the B,C coordinates.

Test8

The paper written by Jonathan Cauldwell, the author takes a stepped approach to creating a Spectrum game in assembly. The first example is a simple "print to the screen". In his example, he sets the A register to 2 and then calls OPENCH (open channel). Channel number 2 is the screen, so he is opening up a channel to the screen. I have not digged deep enough to know much about channels, and the program crashed on the T/S 2068. In reading the Technical Manual, I found that the Function Dispatch version of OPENCH expects to have the channel on the calculator stack. I found the ROM routine STACKA, that put the contents of the A register onto the calculator stack. This still crashed.

In the end I just removed the OPENCH ROM call and the program worked.

The main part of this program is to call PR_STR, or Print String. This call is not documented as a ROM call, but as a subpart of a ROM call. Luckily I found a similar example for the T/S 2068 and it worked.

Test9

This is more of the program from Cauldwell and it sets up a UDG character and has it move up and down the screen in an endless loop. It is a interesting program that really does not use any ROM calls. It is useful in showing how to set up UDG's.

Test10

I found a different example of printing a string, but using the PRINT ROM call. In this example, instead of telling the rouinte how many characters to print, it will print until it finds a specific character or byte.

Test11

This program gets a character from the keyboard and prints it to the screen. It uses the LAST_K (last key) system variable to find out what key the user touched. The system variable is set to 0 and when it changes, the program knows to print the character.

```
;; test11.asm
;;     Print characters typed on the
keyboard

ORG 65000

start:
          LD HL,23560
     ;  Put LAST_K into HL
          LD (HL),0
     ;  Let LAST_K to 0
loop:     LD A,(HL)
     ;  Let A = LAST_K
          CP 0
     ;  is LAST_K still 0?
          JP Z,loop
     ;  If it is, loop
          RST 16
     ;  If not, then print the
character
          JP start
     ;  Go back to the beginning

END 65000
```

On the ZX81, getting a character from the keyboard was a two step process in getting the keyboard input and then converting that to a character value. With the T/S 2068 and this method, it is a single step. Since the system variables are basically the same between the Spectrum and the T/S 2068, this program would work on the Spectrum.

Loading Screens

I was recently reading a story about an old Apple II interactive fiction game called "Time Zone". It came out in 1982 and was composed of 6 floppy disks. In those days, a game would normally take only a single disk, so this game was quite large.

In thinking about the game, I thought about how the graphics were probably saved on disk and just loaded to memory to display them. This allowed the game to have an almost unlimted amount of screens.

I was thinking about how you can't do this with the ZX81, but then I realized that you could, well, at least with the sz81 emulator.

In the last issue I wrote on how sz81 has a new LOAD command that will load data from disk to memory. If the data on disk is a dump of screen memory, then when loaded to the ZX81, you could put it on the screen.

To test this, I knew that I had a few things to do. I first had to create a demo ZX81 screen in a binary file on my OS. Once that was created, I could load the file to memory, just like I had previously, to high memory. To get it to screen memory, a short assembly language routine could be written. Then a bit of BASIC would put all of this together.

It was fairly simple to write a C program that would write out the value 136 ($88) to disk (this is the grey square) 32 times, and then a 118 ($76). This was done 22 times. I did not want the extra 2 lines in the lower area of the screen.

The assembly routine came from Toni Baker's machine code book in Chapter 12, where she talks about screens. The assembly is a simple block move from one memory location to another:

```
    LD HL,32770
    LD DE,(D_FILE)
    LD BC,726
```

```
        LDIR
        RET
```

The BASIC code to connect this together is fairly simple:

```
REM XXXXXXXXXXXXXXXX
LOAD "TEST.SCR; 32770"
RAND USR 16514
```

I used zxtext2p to create the .P file for the BASIC program. I used Pasmo to compile the assembly language routine into a .bin file. Using 'dd' I added the assembly language binary to the .P file (see the last issue).

From there I loaded up sz81 and ran the program. I had a few minor mistakes in the assembly langauge code and one mistake in the BASIC code, but with those fixed, the screen was soon filled with grey squares.

Looking more into this, I realized that I did not need to load the file into higher memory, but could load it straight into screen memory. Here is the BASIC program to do this:

```
LET X = PEEK 16396 + 256 * PEEK 16397
LET X = X + 1
LET A$ = "TEST.SCR; " + STR$ X
LOAD A$
```

The first parts gets the address of screen memory from the D_FILE system variable. I incremented X because I did not create a leading $76 in the file, as needed for screen memory. I then created a string with the LOAD option and converting X to a string to be part of it. Then LOAD takes the string and loads the file.

This procedure has the feature of not taking up any ZX81 memory, other than screen memory. A large number of screens can be put on disk and then loaded over and over, taking up no additional memory.

The screen.bas program demonstrates screen loading using the first method. The second example, screen2.bas, loads three example screens, one after another. There is a pause between screens, so hit any key to move to the rest. The screens are test.scr, test1.scr and test2.scr.

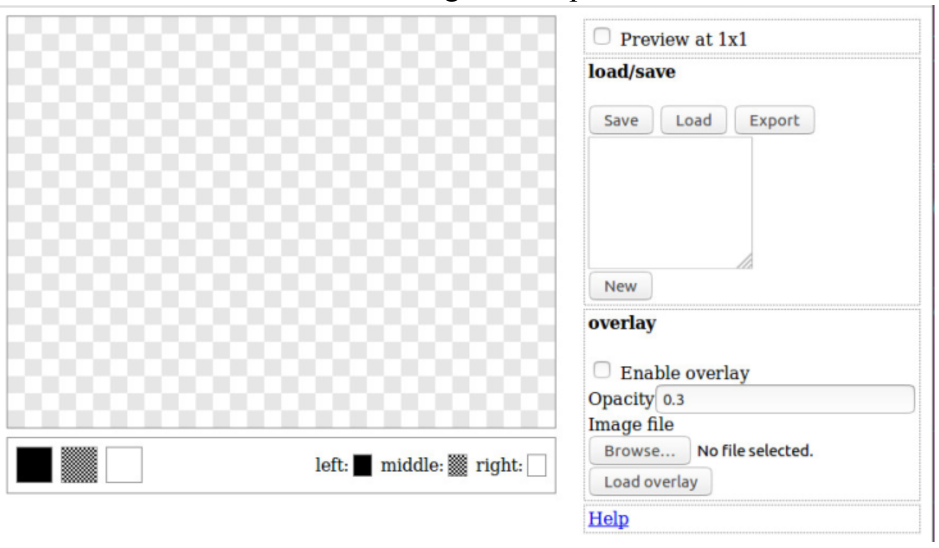
If creating an interactive fiction game, the text and logic of the game could take up as much memory as possible, with the game screens all coming from disk. The screen loaded does not need to be a full screen, so that there is room for the text of the game. The LOAD command will load in as much of the screen that is in the file. If the file only holds half a screen, then it will load half a screen.

ZXpaintyOne

After tinkering with loading screens from disk, I was next thinking about how to draw the screens. ZXpaintyOne is a javascript program that runs in a web browser that works like a version MS Paint, but only with the character graphics of the ZX81.

When ZXpaintyOne is started, there is a blank drawing area on the left. The background is a light hex pattern that shows the character blocks on the ZX81 screen. The drawing area is a full 24 lines, which includes the two lines reserved for BASIC input.

Using the left mouse button, black pixels are drawn on the screen. If the cursor is moved too fast, the pixels will skip and not draw a complete line, circle, arc, etc. The right mouse button is essentially an erase button, but one would also look at it as drawing a white pixel on the screen. The



middle button handles the gray characters. The smallest gray in the ZX81 character set is a half character (or about two pixels wide).

Once the picture is drawn, there are two ways to save the image. It is possible to create a .P file that creates a BASIC program with PRINT statements to create the image on the ZX81. With this option, it is not possible to convert the .P file back into the image in ZXpaintyOne.

The other option is to "save" the image. A small window just under the Save button is then filled with the hex code for the value of the characters in the image. The user then puts the mouse cursor in this windows and selects all of the text, where it can be copied and then pasted into a text file. Later the characters from the text file can be copied and pasted back into the window in ZXpaintyOne. Then the Load button is clicked and the image is back into the drawing screen, ready to be edited.

This is all great, but I needed to get the image I had drawn into a format that can be loaded into the ZX81. Looking at the characters in the Load/Save window, I noticed that they are the hex value of the character in that part of the screen. The string of characters does not include anything other than what is drawn. Each space on the screen is represented with two characters. I needed a way to convert this string into a binary file.

With a little C coding, I was able to get what I needed. The C program takes the text file, reads in two characters at a time and coverts that to a decimal number. This decimal number is then written to an output file.

The ZX81 screen layout has 32 characters then a $76 which acts as an end of line marker. The text file did not include an $76, so the C program has to read in 32 characters, output 32 characters, and then output a $76. I do not plan on using the lower 2 lines of the screen, my program only reads 22 lines.

The resultant binary file can be loaded directly into the ZX81. After debugging the C program and using the Linux od command to check that my output is what I wanted, loading the screen into sz81 worked the first time.

The program is run like this:

```
 zxpainty infile.bin outfile.scr
```

When I create the file for the hex values from ZXpaintyone, I give the file an extension of .bin (short for binary). When the file is turned into the format for the ZX81 screen, I use the .scr extension. The program requires an input file name and an output file name, in that order. If one of the files can not be opened the program will exit, letting you know which file could not be opened.

The C program is fairly simple and should be able to be compiled for almost any environment.

QDDASM - Quick and Dirty DisAssembler

I've been tinkering with Z80 disassemblers and the ones that I found had some issues. I could not get the disassemblers to output the right information. I realized that to get what I wanted, I would have to write my own disassembler.

QDDASM is a disassembler for the Z80 written in portable C code. QDDASM stands for Quick and Dirty Disassembler. Instead of using a table to convert numerical values to an opcode, I decided to keep it simple and use a number of IF statements. It makes the code long, but it should be fairly easy to understand what the disassembler is doing.

The code is simple and portable. It should be easy to compile on most platforms. Despite the large number of IF statements, I found that the program runs fast on my Linux laptop. Disassembling a large program took almost no time.

QDDASM was written specifically for the ZX81 and has a number of ROM locations built in, so when it sees a CALL statement it can print the right ROM location when it finds it. If additional ROM routines need to be added it is fairly simple to add them, the end user just needs to know the decimal value of the location of the ROM routine. Even for someone that is unfamiliar with C should be able to see the existing code and add to it. Recompiling is simple:

```
% cc -o qddasm qddasm.c
```

qddasm will take a binary file, reading the bytes and converting to Z80 assembler code.

Usage:

qddasm file pc bytes

file - the name of the file to disassemble.
pc - the start of the program counter, or what value or memory location should qddasm list for the first opcode disassembled.
bytes - the number of bytes to read in the file before starting the disassembly.

Dissassembly a ZX81 .P file

The usual way that assembly programs are done on the ZX81 is to put the machine code into a REM statement at the beginning of the program. Usually a call is made to location 16514 to start the assembly part of the program. QDDASM is designed to work in these situations.

hello1.p is an included file. It was created from the assembly language file hello.asm by the PASMO assembler. This will be the first example file to show how to use the disassembler.

Here is the orignal assembly for hello1.p:

```
        LD      HL,line
PLINE   LD      A,(HL)
        CP      $FF
        JP      Z,ENDD
        CALL    PRINT
        INC     HL
        JP      PLINE
ENDD    RET

line:  DEFB
_H,_E,_L,_L,_O,$00,_W,_O,_R,_L,_D,
$76,$ff
```

When disassembling the .P file, the first step is to make sure that there is a REM statement at the beginning of the program. Use the included program, ZX81LIST to list the BASIC lines in the program. This program is from Jack Raat. Run the program like this:

```
%  zx81list hello1.p hello1.out
%  cat hello1.out

----- START OF LISTING -----

   1 REM
   1 RAND USR VAL "16514"
```

This will output the file, hello1.out, to the screen. Notice that line 1 is a REM statement.

The next step is to find location of the REM statement in the .P file. The program, find_rem, reads through the .P file and locates the position of the REM statement. Run it like this:

```
%  find_rem hello1.p
    REM is at 121
```

The output shows that the REM statement is the 121st byte in the program.

Now that we know where to start the disassembler, run QDDASM like this:

```
%  qddasm hello1.p 16514 121
```

This tells qddasm to open hello1.p and to start disassembly at the 122nd byte in the file. In other words, to skip the first 121 characters. It also will start the program counter at 16514, which is the memory location of the first character in the REM statement if it was in ZX81 memory.

Here is the output from qddasm:

```
16514    LD A,(HL)
16515    JP 16521
16518    NOP
16519    NOP
16520    HALT
16521    LD HL,16538
16524    LD A,(HL)
16525    CP 255
16527    JP Z,16537
16530    CALL PRINT
```

```
16533    INC HL
16534    JP 16524
16537    RET
16538    DEC L
16539    LD HL,(12593)
16542    INC (HL)
```

Note that the first bit of assembly is different than the original code. There is an include file that I use when creating a .P file in PASMO that comprises the first 5 bytes. The main thing to notice is that it does a JP to 16521, which is the start of the original code.

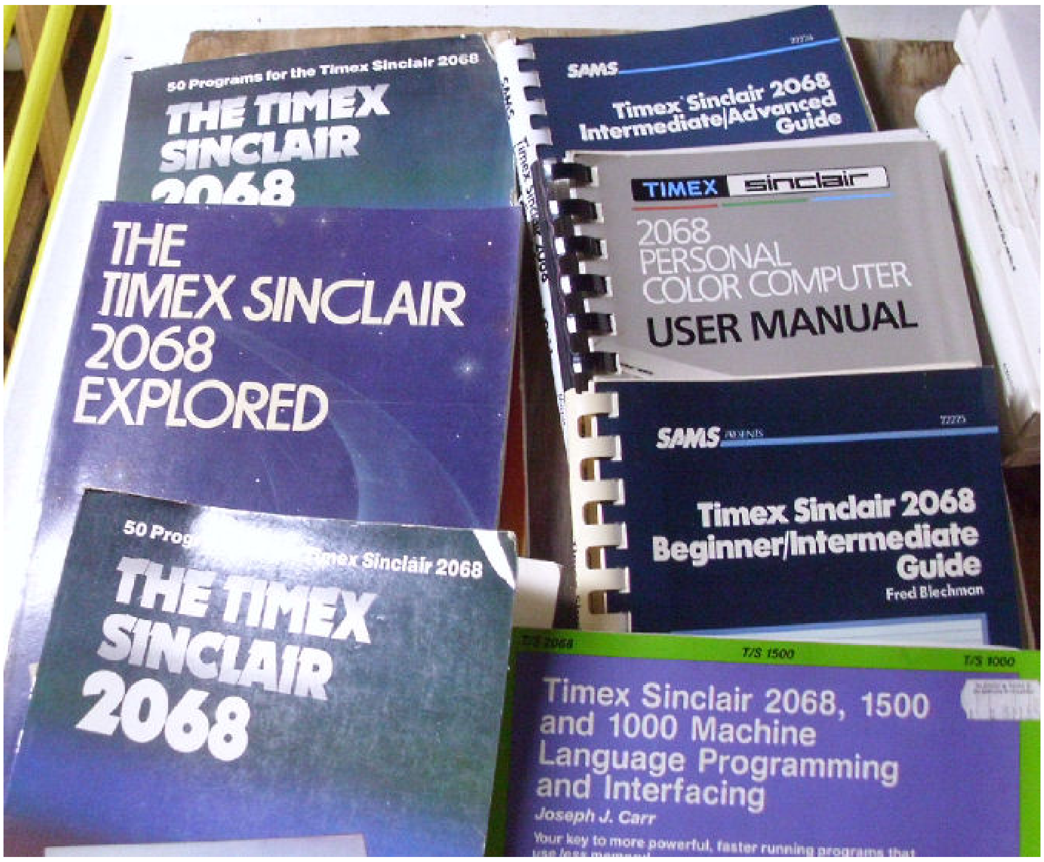
Another note, all output from qddasm is in decimal and not hexadecimal, as in the original code.

Also note that after the RET in line 16537, there is a DEC L, which is not part of the original code. Qddasm does not know of a byte read in is an opcode or data and it assumes that it is an opcode. If the assembly program puts all of the data at the end of the code (like the hello1.asm example) then the disassembly should be accurate. If the data is embedded in the assembly, then the disassembly could be inaccurate.

Limitations of QDDASM

The biggest issue with QDDASM is that is not possible for the disassembler to know if a binary value is an opcode or a bit of data. The disassembler will always assume an opcode. If there is data in the middle of the disassembly, then this could lead to inaccurate results, esp. of the binary value is also an opcode that needs to read in additional data.

I've looked into using QDDASM for disassembly of the T/S 2068 ROM. I first started with a Spectrum ROM and found in the published Spectrum ROM disassembly that there are some unused memory locations on ROM that QDDASM will treat as an opcode, leading to inaccurate results. I might work on a solution so that QDDASM can work on disassembling a ROM.


50 Programs for the Timex Sinclair 2068
THE TIMEX SINCLAIR 2068
SAMS
Timex Sinclair 2068 Intermediate/Advanced Guide
TIMEX sinclair
2068 PERSONAL COLOR COMPUTER USER MANUAL
THE TIMEX SINCLAIR 2068 EXPLORED
SAMS
Timex Sinclair 2068 Beginner/Intermediate Guide
Fred Blechman
50 Programs for the Timex Sinclair 2068
THE TIMEX SINCLAIR 2068
T/S 2068
T/S 1500
T/S 1000
Timex Sinclair 2068, 1500 and 1000 Machine Language Programming and Interfacing
Joseph J. Carr